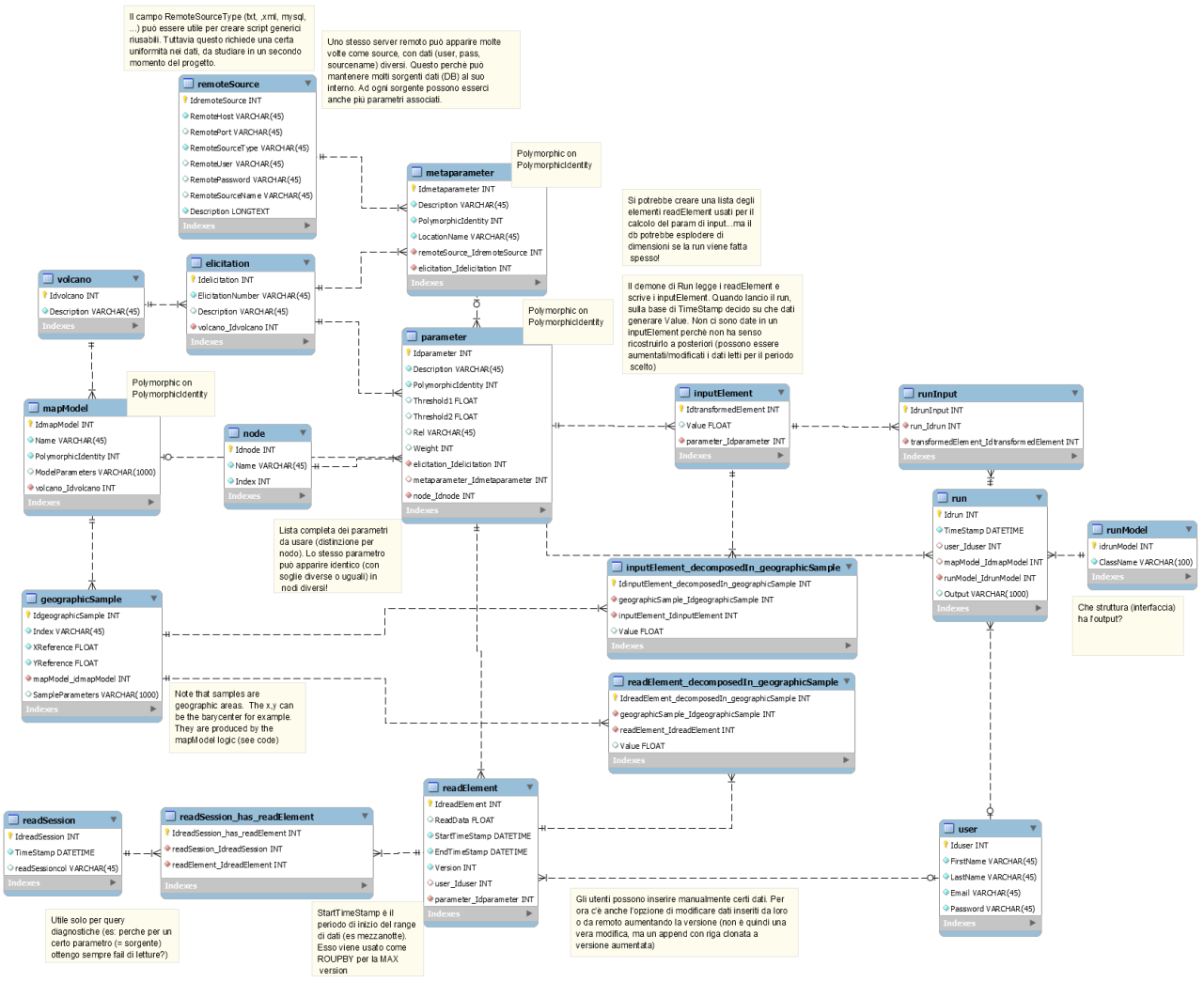
PROBLEMA E ANALISI
di Marco Cincini

A partire dalle descrizioni fornite si è costruito un sistema che rispetta e risolve la seguente descrizione del problema. Tra parentesi, sottolineati, i concetti che corrispondono a tabelle del database.

"Il sistema serve ad ottenere da molteplici database, anche dislocati in sedi remote rispetto il luogo di installazione, dati tempo-varianti necessari all'esecuzione di modelli di calcolo atti a monitorare (e prevedere) l'attività vulcanica dei Campi Flegrei (ed idealmente di qualsiasi altro sito vulcanico).
La tipologia di dati è definita a priori a seguito di un processo noto come elicitazione (elicitation), in cui un insieme di esperti concorda sull' insieme di parametri (parameter) da utilizzare per monitorare un certo vulcano (volcano). Ogni parametro è caratterizzato da una descrizione, due soglie, una relazione, un peso ed un indice univoco. I parametri sono raggruppati in nodi; si noti che posso esserci presenti due parametri identici ma in nodi differenti. Si vuole mantenere uno storico dei parametri e metaparametri usati durante le elicitazioni passate per un certo vulcano; i dati di questi e la relativa logica non dovranno quindi essere sovrascritti.
Data la lista completa di parametri per l'elicitazione corrente, il sistema deve poter svolgere la "fetch" di tutti i valori di questi ad un certo intervallo temporale, ciascuno dalla relativa fonte di dati, anche remota (remoteSource). Per alcuni di questi parametri, voglio inoltre ottenere, non solo un singolo valore ad un certo istante,ma anche la distribuzione geografica di tal valore su una mappa (mapModel) suddivisa in campioni geografici (geographicSample) di forma e numero qualsiasi ("nodo4" del processo di elicitazione). Questa distribuzione si può ottenere solamente se il database di origine contiene riferimenti geografici. Siccome esistono varie tipologie di mappe (a griglia, a cerchi concentrici, da shapefile,..) quando viene svolta la fetch deve essere possibile decidere a quale mappa riferirsi per il calcolo della distribuzione . La tipologia di database su cui sono presenti i valori può variare, è necessario quindi non fare supposizioni sulla tecnologia.
Scorrendo le liste di parametri si può notare come molti di essi facciano riferimento ad uno stesso concetto; ad esempio "Uplift cumulativo ultimi 3 mesi", "Rateo Uplift", ..., fanno tutti riferimento al concetto di Uplift, anche se trattato in modo differente. Questi parametri faranno quindi riferimento agli stessi dati di origine, questa feature si può sfruttare per minimizzare il numero di fetch; dati n parametri basati su uplift posso svolgere infatti una singola fetch (in questo caso uplift prende il nome di metaparametro (metaparameter)).
I dati in input sono grezzi e devono venire elaborati dal metodo di fetch ad uno "standard" definito sulla base del singolo parametro; ad esempio si vuole calcolare la media o il massimo tra tutti i valori dell'ultima giornata (a partire dalle 00:00 all'ora corrente). Un dato cosi calcolato, definito elemento letto (readElement), è costituito da un valore, dall'ora di partenza e fine (ora corrente o 23:59:59 nel caso esegui la lettura in una giornata successiva), da una versione del dato. L'elemento così composto può essere salvato nel database locale. Si noti che la versione è utile perchè, se si eseguono in momenti diversi molteplici fetch relative ad una stessa giornata, il valore calcolato può cambiare. E' quindi necessario tenere traccia del dato più recente. Oltre l'orizzonte della singola giornata già introdotto, deve essere possibile settare come "blocco" temporale per il calcolo anche l'ora, il minuto (per i parametri in cui si necessita un'alta risoluzione) o multipli di questi.
Per i parametri per i quali è stato deciso di calcolare anche la distribuzione geografica, il relativo readElement corrente deve venire "scomposto" anche in sotto-elementi relativi ognuno ad un campione geografico (readElement_decomposedIn_geographicSample).
La procedura di fetch introdotta è da pensarsi completamente automatizzata; essa viene lanciata con cadenza regolare. Durante la prima installazione, si deve pensare di essere in grado di fetchare anche tutti i dati storici precedenti al giorno corrente. Si noti che per alcuni parametri però al momento non esistono fonti di dati a cui accedere per determinarne in maniera automatica l'evoluzione del valore. Per ovviare a questo problema un operatore autorizzato (user) deve inserire/modificare manualmente tale dato (già in forma "standard" come introdotto in precedenza).
I valori letti e salvati nel database locale ora sono in forma adeguata per essere presi in input da un servizio di "run". Si noti che quasi mai, nella descrizione dei parametri, si fa riferimento al dato relativo alla singola giornata (ora, minuto) ma piuttosto ne viene usata una versione "inerziata" su un arco temporale più lungo. Per questo motivo, per ogni parametro, non si da in input al modello di calcolo solamente il singolo readElement (e relativi valori decomposti geograficamente su mappa), ma un valore calcolato sulla base di tutti i readElement relativi all'arco temporale inerziale (per ogni "blocco" di giorno/ora/minuto si dovrà prendere il dato a maggior versione se ne sono presenti più di uno). Tale valore prende il nome di inputElement e può essere ad esempio la media dei readElement degli ultimi 90 giorni. Stessa procedura si applica ai valori decomposti geograficamente (inputElement_decomposedIn_geographicSample). Anche questi valori di input dovranno essere salvati nel db locale in quanto voglio poter essere in grado in futuro di capire quali dati hanno prodotto un certo risultato.
Tutti gli elementi di input prodotti vengono passati al modello (runModel) attraverso un'interfaccia predefinita. Si noti che possono esistere più modelli. Il lancio ad un determinato istante, (run), produrrà, attraverso il modello, un determinato output. Le caratteristiche dell'output non sono note a priori perchè dipenderanno dal modello scelto. A livello di database deve esser possibile mantenere lo storico delle run (e relativi input, come già specificato). L'output,

oltre che essere salvato, dovrà essere anche visualizzabile attraverso browser.
Come per il servizio di fetch, anche il servizio di run sarà svolto in automatico, con la possibilità di essere invocato manualmente da un utente autenticato."



DATABASE E MAPPATURA TRA TABELLE E CLASSI

Schema del database

Lo schema qui riportato, definito per MySQL, (ov.mwb), indica fedelmente tutte le tabelle e relazioni (anche se le variabili possono avere nomi leggermente differenti). Tuttavia esso è solo di natura illustrativa, il programma svolto in python è infatti in grado di costruire la struttura del database da zero (grazie a SQLAlchemy), senza alcun progetto sotto. Modifiche ad esso non corrispondono modifiche nel db reale. Per modificare la struttura del database è necessario alterare e lanciare direttamente il codice (vedere paragrafo "Codice per generazione automatica dello schema"). In particolare tutte le tabelle riportate sono presenti nello script ov_model.py e sono mappate in classi. Le tabelle "polimorfiche" corrispondono a superclassi, la cui discendenza è presente in altri script: ov_model_remote_metaparamer.py e ov_model_concrete_metaparamer.py per la classe metaparameter (due livelli di discendenza), ov_model_concrete_paramer.py per la classe parameter, ov_model_map.py per la classe mapModel.
I concetti principali legati alle tabelle verranno dettagliati nei prossimi paragrafi.

Operazioni preliminari (costruzione e popolazione)

Di seguito vengono dettagliati alcune operazioni da svolgere preliminarmente al fine di garantire la corretta esecuzione del programma. Essendo al momento il programma sprovvisto di un'interfaccia grafica è necessario lanciare uno script di inizializzazione (ov_service_initialization_main.py) per genera la struttura del db (in automatico) e riempirne alcune tabelle con dei valori di test. Lo script esegue senza errori solamente se il db specificato nel main è inesistente. Di seguito viene riportata la lista delle tabelle riempite dallo script e alcuni esempi.

Volcano**:**

- Esempio: `Description: "Campi_Flegrei")`

Elicitation:

- Esempio: `ElicitationNumber: "1", volcano_Idvolcano: "(chiave esterna all'Id del vulcano a cui questa elicitazione è riferita)"`

RemoteSource:

- Esempio: `RemoteHost: "localhost", RemotePort: "3306", RemoteSourceType: "mysql", RemoteUser: "root", RemotePassword: "root", RemoteSourceName: "observationsTest"`
- RemoteSource identifica uno schema (db), “observationsTest”, all'interno di una macchina server “localhost”.
- la connessione ai db è svolta tramite SQLAlchemy, esso è in teoria in grado di connettersi a vari tipi di db con il codice già scritto; tuttavia test sono stati svolti sono con mysql, per attivare altre tipologie di dbms modificare _supportedDbs in ov_service_fetch_main.py. Si veda paragrafo "Implementazione del servizio di Fetch" per maggiori dettagli.
- Nota: i dati scritti in questa tabella devono corrispondere a quelli di database realmente esistenti! Per creare tali database, lanciare lo script di inizializzazione settando la variabile "createTestDatabases = True" (già impostata in questo modo di default).

Metaparameter:

- Esempio: `Description: "metauplift", PolymorphicIdentity: "2", remoteSource_IdremoteSource: "(chiave esterna all'ID della fonte dati)", LocationName: "simulatedObservation1", elicitation_Idelicitation: "(chiave esterna all'Id dell'elicitazione a cui questo metaparametro è relativo)"`
- Il campo “PolymorphicIdentity” deve essere un valore numerico differente per ogni riga inserita relativa alla stessa elicitazione; esso servirà nel codice per identificare quale è la classe relativa contenente la logica di mappatura del metaparametro nella fonte remota. Cambiando elicitazione lo stesso metaparametro può riavere lo stesso PolymorphicIdentity (entrambe le righe saranno quindi relative ad una stessa classe, sono disposti degli switch per gestire le eventuali differenze di comportamento). Per maggiori dettagli vedere paragrafo "Mappatura tabelle contenenti dati righe polimorfiche".
- Più metaparametri possono fare riferimento alla stessa RemoteSource in quanto i metaparametri possono provenire dallo stesso schema (db). Il campo "LocationName" specifica il nome della tabella da cui i valori del metaparametro vengono estratti.

Node:

- Esempio: `Name: “Unrest”, Index: “1”`
- Dall'elicitiazione si evince che i nodi a cui appartengono i parametri dovrebbero essere tre: unrest, magmatic, eurption.

Parameter:

- Esempio: `Description: "upliftthreemonths", PolymorphicIdentity: "3", Threshold1: “2”, Threshold2: “5”, Rel: “<”, Weight: “1” metaparameter_Idmetaparameter: "(chiave esterna all'ID del metaparametro)", elicitation_Idelicitation: "(chiave esterna all'Id dell'elicitazione a cui questo parametro è relativo)", node_IdNone: “(chiave esterna all'Id del nodo)”`
- Il campo “PolymorphicIdentity” deve essere un valore numerico differente per ogni riga inserita relativa alla stessa coppia (elicitazione, nodo); esso servirà nel codice per identificare quale è la classe relativa contenente la logica di calcolo del parametro.
- Cambiando elicitazione e/o nodo, lo stesso parametro può riavere lo stesso

PolymorphicIdentity (entrambe le righe saranno quindi relative ad una stessa classe); questo perchè tal parametro potrebbe ripetersi, anche con stessi valori, in un nodo differente all'interno della stessa elicitazione. Non è ammessa invece duplicazione di un parametro all'interno di un nodo della stessa elicitazione.

RunModel:

- Esempio: `ClassName: "TestModel"`
- Il nome inserito deve corrispondere con il nome di una classe (che conterrà lo script relativo al modello di calcolo) nella sottocartella runmodels. Si veda capitolo codice per maggiori dettagli.

MapModel e GeographicSample:

- Per comprendere la semantica dei dati generati si rimanda a paragrafo "Generazione automatica di una mappa"

Come già introdotto, oltre la costruzione e popolazione del database locale, lo script inizializzazione genere, se settato con "createTestDatabases = True" anche due database di test (sempre in ambiente locale). Senza questi database ovviamente i servizi che verranno di seguito introdotti non possono funzionare in quanto mancherebbero i dati di origine. Generando tali database è garantito che l'esecuzione dei servizi proposti (fetch, run) avrà successo e produrrà dati analoghi a quelli presentati negli esempi di questa relazione.
Per maggiori dettagli sul contenuto e sull'implementazione dei database di test si rimanda al paragrafo "Simulatore di test, un tool per lo sviluppo".

Codice per generazione automatica dello schema

Al fine di generare automaticamente lo schema presentato è necessario definire delle classi derivanti da un oggetto "Base" relativo a SQLAlchemy, creato in ov_model.py con la seguente sintassi:

```
Base = declarative_base()

class ReadSession(Base):
    __tablename__ = 'readSession'
    _idReadSession = Column('idreadSession', Integer, primary_key=True)
    _timeStamp = Column('timeStamp', DateTime, default=datetime.datetime.now, nullable=False)

    _readSession_Has_ReadElements = relationship("ReadSession_Has_ReadElement",
                                                 backref="readSession")
```

Dalla sintassi si evince che la classe "ReadSession" presentata è mappata nella tabella "readSession". Oltre la definizione dei due campi/colonne (idreadSession, timeStamp) è riportato anche un campo definito attraverso una relationship. Questo indica che gli oggetti di tipo ReadSession avranno associati una lista di oggetti di tipo ReadSession_Has_ReadElement (questa tipologia di sintassi è assente nei database, li si ha infatti solo il riferimento inverso definito dalle chiavi esterne).
Come regola generale si sono utilizzate delle chiavi primarie surrogate, cioè non dipendenti dalla semantica della tabella. La chiave primaria, in ogni caso, è sempre una singola colonna di interi crescenti.
Si rimanda al file db_manager.py per la descrizione del codice relativo a connessione e creazione (oltre che gestioni transazionali di sessioni di comunicazione) di un database attraverso SqlAlchemy.

Mappatura delle tabelle contenenti righe polimorfiche

Tre delle tabelle presenti nello schema contengono una colonna, PolymorphicIdentity, che viene

utilizzata da SqlAlchemy per mappare le righe in classi differenti a seconda del valore di esso. Queste tabelle sono MapModel, Metaparameter, Parameter. Le relative classi base sono contenute tutte in ov_model.py, come per le classi relative alle altre tabelle. Nelle classi base deve venir specificato il seguente costrutto:

```
class MapModel(Base):
    # ...
    # sqlalchemystandard way to define a column
    _polymorphicIdentity = Column('polymorphicIdentity', Integer, nullable=False)
    # ... more columns definition...

    # sqlalchemy single table inheritance
    __mapper_args__ = {
        'polymorphic_on': _polymorphicIdentity
    }
```

Mentre nelle classi derivate, presenti rispettivamente in: ov_model_remote_metaparamer.py, ov_model_concrete_metaparamer.py, ov_model_concrete_paramer.py, ov_model_map.py, va specificato il seguente codice:

```
class GridModel(MapModel):

    __mapper_args__ = {
        'polymorphic_identity': 1
    }
```

Ogni sottoclasse relativa alla stessa classe padre dovrà avere necessariamente un valore diverso. Si è scelto di non assegnare alcun valore alle classi padre in quanto esse sono considerate non istanziabili.
Quando vengono svolte delle query, SqlAlchemy è in grado di associare ad ogni riga caricata proveniente da tabelle polimorfiche, la classe relativa. La lista tornata dalla query sarà quindi composta da oggetti di tipo diverso (con stesso supertipo).
La metodologia presentata prende il nome di "single table inheritance". Documentazione: http://docs.sqlalchemy.org/en/improve_toc/orm/inheritance.html#single-table-inheritance

STRUTTURA DEL CODICE E FUNZIONALITA'

Struttura generale

Il codice implementato è strutturato nella seguente maniera. Oltre le classi "di modello", già presentate, che hanno una mappatura nel database, il codice è composto da servizi, ossia funzioni complesse che hanno lo scopo di mappare la dinamica del problema presentato. In particolare sono presenti:

- MapModelCreationService (file ov_service_initialization_main.py). Servizio di supporto da lanciare durante l'inizializzazione per generare (e salvare) delle mappe a partire da parametri in input.
- FetchService (file ov_service_fetch_main.py). Servizio in grado di leggere i dati sulle fonti remote, elaborarli in uno standard definito (aggregazione dati ed eventuale distribuzione di questi in relazione ad una mappa), e scriverli sul database locale.
- RunModelService (file ov_service_runmodel_main.py). Servizio che prende in input i dati già standardizzati dal database locale e li passa ad un RunModel (scritto da uno scienziato) al fine di ottenere un output.

Ognuno di questi servizi è legato ad un main (nello stesso file) in quanto essi rappresentano funzionalità distinte da eseguire in maniera indipendente eventualmente da processi separati.

Esistono poi una serie di altre classi di supporto che verranno introdotte durante la spiegazione dei vari servizi.

Generazione automatica di una mappa (MapModelCreationService)

Con mappa si intende una struttura "virtuale" che suddivide un territorio in aree predeterminate (campioni). La suddivisione è molto utile per creare modelli di calcolo precisi che sfruttano parametri geolocalizzati; per questi, infatti, un certo valore è caratterizzato non solo dall'istante in cui avviene, ma anche da coordinate X e Y.
L'obiettivo ultimo della mappa è quindi quello di sapere, per un certo periodo temporale, la distribuzione dei valori (medi, massimi, ..) di un certo parametro all'interno dell'area oltre che il suo valore aggregato "globale".
Le mappe si distinguono per forma e numero di campioni; il software è stato costruito in modo da poter essere facilmente espanso in modo da integrare nuove tipologie di mappe. La tabella MapModel è composta dai campi: Name, ModelParameters (opzionale, un lungo campo stringa riempibile con valori custom separati da virgola a seconda del tipo di mappa; utile per fornire parametri geometrici globali), PolymorphicIdentity, vulcano di riferimento. La tabella GeographicSample invece è composta da: Index (valore univoco di un campione all'interno di una mappa), XReference e YReference (coordinate del campione, il sistema di riferimento usato dipende dal tipo di mappa), SampleParameters (opzionale, un lungo campo stringa riempibile con valori custom separati da virgola a seconda del tipo di mappa; utile per fornire parametri geometrici relativi al solo campione), mappa di riferimento.
Attualmente lo script di inizializzazione (ov_service_initialization_main.py) crea in modo automatico due mappe di esempio (attraverso il servizio MapModelCreationService) sulla base di parametri impostabili; una a griglia rettangolare ed una chiamata modello a "sezioni cardinali". Le classi che contengono il comportamento relativo a queste due di mappe sono in ov_mopdel_map.py (classe GridModel, mappabile sul Db tramite PolymorphicIdentity = 1 e classe CardinalSectionsModel, mappabile tramite PolymorphicIdentity = 2). Di seguito ne vengono riportate le caratteristiche.

GridModel:

- 4 parametri globali: xResolution, yResolution, xSize, ySize.
- I campioni sono distribuiti a griglia e separati tra loro di xResolution unità lungo X e yResolution unità lungo Y.
- I campioni non hanno associati parametri (la loro dimensione è fissa e stimabile a partire dai soli dati globali).
- Sistema di riferimento centrato nel corner left, bottom; X crescente fino a xSize verso destra, Y crescente fino a ySize verso l'alto.
- La localizzazione di un punto "query" all'interno della griglia avviene sfruttando la conoscenza geometrica del modello e non richiede il confronto punto per punto.

CardinalSectionsModel:

- Struttura geometrica costituita da due cerchi concentrici. Quello interno è inteso come bocca del vulcano. Quello esterno è suddiviso da 4 segmenti diagonali in 4 parti N,S,W,E, che rappresentano i 4 versanti del vulcano.
- 2 parametri globali: innerCircleRadius, outerCircleRadius.
- I campioni sono fissati a 5, il primo (Index = 0) centrato nel cerchio interno, ed i rimanenti 4 disposti nelle aree esterne.
- Sistema di riferimento centrato nel campione con Index = 0. X crescente verso destra, Y crescente verso l'altro.
- I campioni non hanno associati parametri (la loro dimensione è fissa e stimabile a partire dai

soli dati globali).

- La localizzazione di un punto "query" avviene sfruttando la conoscenza geometrica del modello e non richiede il confronto punto per punto. I punti che "cadono" fuori dal cerchio più esterno sono considerati fuori dalla mappa.

Oltre a queste è fornita una terza classe, ShapeFileModel, di cui è stata definita però solamente l'interfaccia. In questo caso l'unico parametro globale è il path allo ShapeFile, mentre i singoli campioni geografici avranno associati parametri relativi ai corner che li distinguono, essendo la loro forma non predeterminata.
In generale è possibile costruire nuove tipologie di mappe implementando una classe che rispetti le seguenti regole: discendenza da classe generica MapModel, associazione con un nuovo valore di PolymorphicIdentity, implementazione della seguente interfaccia (presa dalla superclasse MapModel):

```
#localize function = x,y must be in a normalized reference system, x and y between [0,1] (x growing
left to right, y growing bottom to top)
def localizePoint(self, x, y):
    raise NotImplementedError("This class is to be considered not-instantiable.")  #check subclasses
(ov_model_map.py)

#private method. Convert a point the current mapModel reference system. x,y must be in a normalized
reference system, x and y between [0,1] (x growing left to right, y growing bottom to top)
def convertPointToMapModelRefSystem(self, x, y):
    raise NotImplementedError("This class is to be considered not-instantiable.")  #check subclasses
(ov_model_map.py)

#will build a new model (with new samples calculated basing on kwargs parameters (defined by the
model itself).
def buildSamples(self, **kwargs):
    raise NotImplementedError("This class is to be considered not-instantiable.")  #check subclasses
(ov_model_map.py)

#check the consistency of the model (gemoteric checks). Used internally to be safe about the points
and parameters loaded from the database.
def checkIntegrity(self):
    raise NotImplementedError("This class is to be considered not-instantiable.")  #check subclasses
(ov_model_map.py)
```

Gestione della logica di metaparametri e parametri

Prima di introdurre i servizi di fetch e run è utile avere una visione chiara di come sono implementati metaparametri e parametri e qual'è la relazione tra essi. Infatti i due servizi utilizzeranno poi estensivamente questi concetti.
L'implementazione è stata svolta in modo da isolare il più possibile le responsabilità degli utilizzatori all'interno di singoli file. In particolare i tecnici informatici hanno responsabilità sui metaparametri e sulle politiche di mappatura di questi su fonte remota (ov_model_concrete_metaparameter.py), gli scienziati che elaborano elicitazione e modello di calcolo hanno invece responsabilità sul calcolo del valore dei parametri (ov_model_concrete_metaparameter.py).
Il primo passo per la gestione dei metaparametri è l'inserimento di questi nel database (operazione manuale), l'esempio fornito come test è illustrato nel paragrafo "Operazioni preliminari".
Per ogni metaparametro inserito nel database con campo PolymorphicIdentity univoco deve venire scritta una classe specifica, di seguito ne viene riportato un esempio completo (trovabile in ov_model_concrete_metaparamer.py) da cui estrapolare l'interfaccia applicabile a tutti i casi possibili.

```
class MetaExample1(RemoteDbMetaparameter):

    __mapper_args__ = {
        'polymorphic_identity': 1
    }
```

Il valore "polymorphic_identity" corrisponde al valore di PolymorphicIdentity nella tabella Metaparameter. E' il campo che permette a SQLAlchemy di eseguire il mapping tra riga e sottoclasse specifica.

```
class Example1RemoteMapped(object): #magically filled by sqlalchemy engine
    #wrap the properties that are interesting
    @property
    def mappedTimeStamp(self):
        return self.timeStamp #column exact name
    @property
    def mappedValue(self):
        return self.value  #column exact name
    @property
    def mappedXLocation(self):
        return self.xlocation  #column exact name
    @property
    def mappedYLocation(self):
        return self.ylocation  #column exact name

    pass
```

La classe interna "Example1RemoteMapped" è la rappresentazione in memoria della tabella remota interessata. La struttura dell'oggetto di questa classe è riempita magicamente da SQLAlchemy; di conseguenza i campi “timeStamp”, “value”, .., sono i nomi delle colonne remote. Le proprietà fornite “wrappano” tali nomi (differenti da caso a caso) in degli alias standard. I 4 campi forniti sono tutti quelli necessari e sufficienti per un parametro geolocalizzato. Nel caso in cui il parametro non sia geolocalizzato sono sufficienti valore e timeStamp.

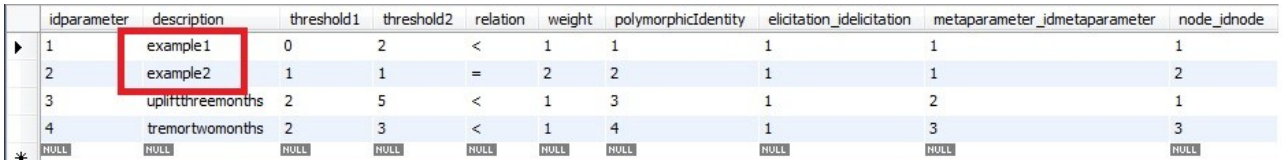
```
def getRemoteClassMapping(self):
    return self.Example1RemoteMapped #passing a class!!! (for queries)

def getRemoteTimeStampMapping(self): #(for queries)
    if self.elicitation.volcano.description == 'Campi_Flegrei':
        if self.elicitation.elicitationNumber == '1':
            return self.getRemoteClassMapping().timeStamp
        else:
            #undefined elicitation
            raise
    else:
        #undefined volcano
        raise
```

"getRemoteClassMapping" e "getRemoteTimeStampMapping" sono due metodi di servizio da riadattare opportunamente. Si può notare l'if/else per gestire i diverse elicitazioni/vulcani.

```
def fillTimeSerie(self, remoteElements):
    ts = TimeSerieGeolocalized()

    #fill elements
    for item in remoteElements:
        value = item.mappedValue
        # here you have to convert to a valid python timestamp if source has different time format!
        t = item.mappedTimeStamp
        # here you have to convert to a normalized and standard reference system if source uses dif-
ferent system -> X(0,1), Y(0,1). (standard: x growing left to right, y growing bottom to top)!
        x = item.mappedXLocation
        y = item.mappedYLocation

        ts.appendSample(value, t, x, y)

    return ts
```

"fillTimeSerie" riceve in ingresso una lista di oggetti di tipo Example1RemoteMapped relativa all'ultima lettura e costruisce una serie temporale dalla lista stessa. Da notare che il database remoto restituisce i dati nel suo formato, quindi non è detto che il valore "mappedTimeStamp" sia un valido oggetto DateTime python. Per questo motivo qui può essere necessario svolgere una conversione dal formato di origine a quello corrente. Questo argomento è valido anche per le coordinate X ed Y; ogni fonte remota ha il suo sistema di riferimento; è necessario quindi qui standardizzarsi ad un unico caso comune, ossia ad un sistema normalizzato con X ed Y comprese tra 0 ed 1 e crescenti

rispettivamente verso destra e verso l'alto. Senza la conversione a tale sistema la localizzazione su mappa dei punti che verrà svolta in un passo successivo non può avere successo. I dati convertiti ed organizzati in una serie temporale sono pronti per essere analizzati nei passi successivi.
Per il caso non geolocalizzato è necessario usare "TimeSerie()" invece di "TimeSerieGeolocalized()".
Il discorso per i parametri è simile a quanto introdotto per i metaparametri.
Nel caso concreto illustrato nella seguente figura sono definiti due parametri "example1" ed "example2" linkati allo stesso metaparametro "metaExample1". Si è rilevato che questa casistica è molto comune; esistono cioè svariati parametri all'interno di un'elicitazione che derivano i loro valori da una stesso concetto, grazie all'introduzione dei metaparametri si riescono quindi a risparmiare varie operazioni di fetch.



idparameter	description	threshold1	threshold2	relation	weight	polymorphicIdentity	elicitation_idelicitation	metaparameter_idmetaparameter	node_idnode
1	example1	0	2	<	1	1	1	1	1
2	example2	1	1	=	2	2	1	1	2
3	upliftthreemonths	2	5	<	1	3	1	2	1
4	tremortwomonths	2	3	<	1	4	1	3	3
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Per ogni parametro inserito nel database con campo PolymorphicIdentity univoco deve venire scritta una classe specifica, di seguito vengono riportati completamente sia "example1" che "example2" (trovabili in ov_model_concrete_paramer.py) da cui estrapolare l'interfaccia applicabile a tutti i casi possibili.

```
class Example1(Parameter):

    __mapper_args__ = {
        'polymorphic_identity': 1
    }

    def __init__(self):
        Parameter.__init__(self)
        self.setTimeSample(Period.DAY, 1)
        self._inertiaDuration = datetime.timedelta(days=90)
```

Nel costruttore del parametro si deve chiamare il metodo "setTimeSample" passando ad esso la durata che si vuole assegnare come intervallo per la fetch. Se ad esempio si setta come valore "1 Day" questo significa che il valore readElement relativo a questo parametro sarà calcolato sulla base di dati proveniente solamente dall'ultima giornata (a partire dalle 00:00). Altri valori sono possibili, ad esempio si può impostare "1 Hour" o "12 Hours"; i valori devono essere sempre tali che scelto X, allora 24 % X = 0 (caso ore). Al momento sono stati effettuati test solamente con valore pari a "1 Day".
Un secondo parametro da settare è "_inertiaDuration", un intervallo temporale che indica su che periodo inerziare i dati letti al fine di produrre valori di input ai modelli di calcolo.
Di seguito vengono mostrati come calcolare readElement e inputElement per il parametro "example1". Per questo parametro di esempio si suppone che esso non sia georeferenziato, cioè non si è interessati (o non si è in grado) a conoscere la decomposizione dei suoi valori sulla mappa.

```
    #returns a readElement
    def calculateValue(self, mapModel, data, startTime, endTime):

        if self.elicitation.volcano.description == 'Campi_Flegrei':
            if self.elicitation.elicitationNumber == '1':

                #calculation of the main value
                total = data.calculateSum()

                return self.initReadElement(None, startTime=startTime, endTime=endTime, value=total)
            else:
                #undefined elicitation
                raise
        else:
```

```
            #undefined volcano
            raise
```

Il metodo "calculateValue", chiamato dal servizio di fetch, prende in input la mappa (opzionale, non necessario in questo caso) ed una serie temporale ("data") contenente già dati pronti per essere analizzati. L'analisi deve essere scritta dallo scienziato responsabile del parametro; se i calcoli da fare non sono complessi si possono già utilizzare i metodi della serie temporale. La serie temporale base (TimeSerie) è in grado di calcolare massimo, minimo, media, presenza (ossia se esiste almeno un valore diverso da 0). Nell'esempio riportato viene quindi svolta la somma dei valori proveniente dall'ultimo giorno. Un'analisi come quella presentata è di tipo aggregato, cioè associa un singolo valore per tutte le rilevazioni dell'ultimo giorno di un certo parametro, le distribuzioni all'interno dei campioni della mappa non sono quindi calcolate.
Non è necessario nemmeno controllare i bound temporali relativi ai dati in quanto è garantito che la serie contiene solamente i dati del corretto range temporale.

```
    #returns a inputelement
    def calculateInertiaValue(self, mapModel, data, geographicDecomposition):

        if self.elicitation.volcano.description == 'Campi_Flegrei':
            if self.elicitation.elicitationNumber == '1':

                #calculation of the main value
                total = data.calculateMax()

                return self.initInputElement(value=total)
            else:
                #undefined elicitation
                raise
        else:
            #undefined volcano
            raise
```

Il metodo "calculateInertiaValue", chiamato dal servizio di run, calcola il valore di input per il modello di calcolo. Esso prende in input la mappa (opzionale, non necessario in questo caso), una serie temporale ("data") contenente 90 valori relativi ai readElement degli ultimi 90 giorni (dal valore di "intertiaDuration") e, opzionalmente, la decomposizione geografica di questi valori aggregati nei singoli campioni della mappa (non presente in questo caso). Nell'esempio riportato viene svolto quindi il calcolo del massimo dei 90 giorni come valore in input al modello.
Di seguito viene riportato un secondo parametro di esempio, "example2", che, diversamente da "example1" calcola non solo i valori aggregati delle tabelle readElement e inputElement ma anche la loro distribuzione data una certa mappa (readElement_decomposedIn_geographicSample e inputElement_decomposedIn_geographicSample). Utilizzando la sintassi dell'elicitazione si può affermare che "example2" è considerato quindi selezionato al nodo4 dell'elicitazione stessa.

```
class Example2(Parameter):

    __mapper_args__ = {
        'polymorphic_identity': 2
    }

    def __init__(self):
        Parameter.__init__(self)
        self.setTimeSample(Period.DAY, 1)
        self._inertiaDuration = datetime.timedelta(days=10)

    #returns a readElement
    def calculateValue(self, mapModel, data, startTime, endTime):

        if self.elicitation.volcano.description == 'Campi_Flegrei':
            if self.elicitation.elicitationNumber == '1':

                #calculation of the main value and the values distributed over the volcano "map".
Only for parameters subselected at node4
                total = data.calculateAvg()
                mapValues = data.calculateAvgOnMapModel(mapModel)
```

```
                return self.initReadElement(None, startTime=startTime, endTime=endTime, value=total,
mapModel=mapModel, geographicDecomposition=mapValues)
            else:
                #undefined elicitation
                raise
        else:
            #undefined volcano
            raise

    #returns a inputelement
    def calculateInertiaValue(self, mapModel, data, geographicDecomposition):

        if self.elicitation.volcano.description == 'Campi_Flegrei':
            if self.elicitation.elicitationNumber == '1':

                #calculation of the main value
                total = data.calculateAvg()

                if geographicDecomposition is not None and isinstance(geographicDecomposition, col-
lections.Iterable):
                    mapValues = []
                    for ts in geographicDecomposition:
                        mapValues.append(ts.calculateAvg())
                else:
                    mapValues = None

                return self.initInputElement(value=total, mapModel=mapModel, geographicDecomposi-
tion=mapValues)
            else:
                #undefined elicitation
                raise
        else:
            #undefined volcano
            raise
```

Nel metodo "calculateValue" la principale differenza è nel fatto che l'oggetto "data" passato in input è di tipo TimeSerieGeolocalized che mette a disposizione metodi supplementari rispetto TimeSerie. In particolare chiamando "CalculateAvgOnMapModel" (o altri metodi analoghi) viene ritornata una lista di N elementi, dove ognuno rappresenta la media del relativo (per indice) campione geografico. Nel metodo "calculateInertiaValue" la principale differenza è nel fatto che il campo "geographicDecomposition" è qui utilizzato e contiene una lista di N serie temporali (TimeSerie), una per campione geografico.

Interfaccia ed effetti del servizio di Fetch (FetchService)

Il servizio di fetch è "pilotato" sulla base di alcune impostazioni che vengono chieste all'utente (ov_service_fetch_main.py). In particolare si deve settare l'Id del metaparametro ed, eventualmente, l'Id del parametro di cui si vuole leggere il valore. Impostando solamente l'Id del metaparametro la fetch costruirà un readElement (e, se possibile, anche dei readElement_decomposedIn_geographicSample legati ad esso) per ogni parametro collegato al metaparametro scelto. In tal modo con una singola fetch si ottengono i valori di più parametri, anche nel caso in cui abbiano settati degli intervalli temporali differenti (1 Day, 1 Hour..). Se invece si imposta sia l'Id del metaparametro che quello del parametro il servizio creerà un readElement per il solo parametro scelto. Nel caso in cui l'Id del parametro non sia dipendente dal metaparametro impostato (vedere tabella "parameter") il servizio ritornerà errore. Quindi, con il seguente codice si esegue la fetch dei parametri "example1" ed "example2" in quanto è stato settato solamente l'Id del metaparametro "metaExample1" (primo attributo).

```
fetchService.runOnLastInterval(1, None)
```

Al contrario, con la seguente chiamata verrà svolta la fetch solamente per "example2".

```
fetchService.runOnLastInterval(1, 2)
```

Per quanto il metodo descritto è l'unico necessario in un sistema consolidato, si è posta l'esigenza di riuscire a svolgere la fetch di tutto lo storico di un parametro e non solo dell'ultimo intervallo (es: ultimo giorno). Questa variante è necessaria per settare il sistema durante la prima installazione.

```
start = datetime.datetime.strptime("2000-1-1 0:0:0", "%Y-%m-%d %H:%M:%S")
end = datetime.datetime.strptime("2015-5-6 0:0:0", "%Y-%m-%d %H:%M:%S")
fetchService.runOnHistoricalData(1, 2, start, end)
```

Il metodo sopra riportato esegue la fetch di "example2" su tutto l'arco temporale che va dal 1 gennaio 2000 al 6 maggio 2015. In questo caso quindi il numero massimo di readElement creati è pari al numero di intervalli contenuti in tale range; nel caso in cui non ci siano valori in alcuni intervalli allora essi saranno nulli (e non verranno scritti sul database).
Altri parametri che è necessario impostare durante la creazione del FetchService sono: vulcano, elicitazione e mappa. Questo perchè, come visto in precedenza, il comportamento del programma è relativo a questi dati (cambiando vulcano è normale ad esempio che i metaparametri siano calcolati da server dislocati in altro luogo).
Nelle seguenti immagini viene riportato il caso di una fetch del solo parametro "example2" su due intervalli passati (30 aprile e 1 maggio). Risultato nella tabella readElement:

idreadElement	readData	startTimeStamp	endTimeStamp	version	user_iduser	parameter_idparameter
1	0.502836	2015-05-01 00:00:00	2015-05-01 23:59:59	1	NULL	2
2	0.491533	2015-04-30 00:00:00	2015-04-30 23:59:59	1	NULL	2
NULL	NULL	NULL	NULL	NULL	NULL	NULL

In questo caso il database di origine contiene dati (random) relativi al 30 aprile e 1 maggio con valori tra 0 e 1 geolocalizzati (la frequenza dei dati è di uno ogni pochi minuti). Come si può notare dal campo "readData" il servizio di fetch ha letto tali valori e calcolato la media (perchè era questo che si era scritto nel codice "calculateValue" del parametro "example2". Dai campi "startTimeStamp" e "endTimeStamp" si riesce bene a cogliere il concetto di intervallo. Si noti che nel caso in cui venga svolta una fetch nel giorno corrente allora il valore "endTimeStamp" sarà pari all'ora corrente! Il campo "user_iduser" non viene mai scritto dal servizio di fetch in quanto è utile solamente per quei parametri considerati manuali, cioè per cui non si ha una fonte remota da cui attingere per il calcolo dei valori. Questi parametri avranno la peculiarità di avere il riferimento a metaparametro nullo. L'inserimento di questi dovrà essere svolto da parte di un operatore (user) tramite un interfaccia grafica. Il campo "version" è di default a 1 quando viene per la prima volta letto un certo intervallo temporale per un parametro. Nel caso la lettura venga ripetuta per tal intervallo (nuova fetch) il servizio scriverà nuove righe con versione aumentata. Questo è necessario perchè non si può supporre a priori che i dati nella fonte remota non cambino.
Essendo il parametro "example2" georeferenziato readElement non è la sola tabella scritta dal servizio. I valori letti relativi ad ogni giornata sono infatti decomposti nei campioni della mappa scelta durante l'inizializzazione del servizio. In questo esempio è stata scelta una mappa di tipo CardinalSectionsModel contenente 5 campioni geografici (si veda paragrafo Generazione automatica di una mappa).

idreadElement_decomposedIn_geographicSample	geographicSample_idgeographicSample	readElement_idreadElement	value
1	701	1	0.485516
2	702	1	0.480929
3	703	1	0.562198
4	704	1	0.514099
5	705	1	0.47
6	701	2	0.50915
7	702	2	0.507347
8	703	2	0.44992
9	704	2	0.5369
10	705	2	0.48935

Dall'immagine si può notare come, nella tabella readElement_decomposedIn_geographicSample, ad ogni readElement corrispondano 5 campioni; per ognuno di esso è calcolata la media. Di seguito, per completezza, vengono anche riportati degli estratti delle tabelle mapModel e geographicSample.

idmapModel	name	polymorphicIdentity	modelParameters	volcano_idvolcano
1	Grid35x20	1	1, 1, 35, 20	1
2	CardinalModelTest	2	10, 20	1

idgeographicSample	index	xReference	yReference	sampleParameters	mapModel_idmapModel
693	692	27	19		1
694	693	28	19		1
695	694	29	19		1
696	695	30	19		1
697	696	31	19		1
698	697	32	19		1
699	698	33	19		1
700	699	34	19		1
701	0	0	0		2
702	1	15	0		2
703	2	0	-15		2
704	3	-15	0		2
705	4	0	15		2

Oltre alle tabelle readElement e readElement_decomposedIn_geographicSample, il servizio di fetch scrive nelle tabelle readSession e readSession_has_readElement. Queste due tabelle servono a tenere un log temporale di quando sono avvenute le letture e quali letture sono state fatte ad ogni sessione.

Implementazione del servizio di Fetch (FetchService)

Di seguito viene riportata una breve descrizione dei passi che compie il servizio di fetch al fine di ottenere i risultati mostrati.

1. InitFetch. Inizializzazione di una nuova readSession all'ora corrente.
2. SearchMetaparameter. Si cerca e si carica in memoria, insieme ad altri dati collegati, il metaparametro con l'Id richiesto.
3. LoadMapModel. Si cerca e si carica in memoria, insieme ad altri dati collegati, la mappa richiesta.
4. EstimateCorrectQueryDateRange / EstimateCorrectHistoricalQueryDateRange. In questo step si cerca di risalire alla data e orario di inizio dell'intervallo temporale minimo tra quelli dei vari parametri scelti per la fetch. Quindi se ad esempio si chiede di fare fetch solamente

per il parametro "example2" a partire dalle 12:00 di oggi, questo metodo sposta indietro il tempo di inizio alle 00:00 sulla base di quanto dichiarato nel parametro (intervallo temporale = 1 Day). Il range complessivo può coprire anche più intervalli temporali (per fetch tramite "runOnHistoricalData", su dati storici).

5. FetchValuesFromSources. Viene creato un oggetto fetcher (files ov_model_generic_fetcher.py e ov_model_db_fetcher.py della sottocartella fetchers). Il tipo dell'oggetto fetcher è stabilito a runtime osservando il campo "RemoteSourceType" della tabella remoteSource legata al metaparametro corrente. Impostando "mysql" l'oggetto creato è di tipo DbFetcher e sfrutta SQLAlchemy per aprire una connessione al database remoto. Attualmente "mysql" è l'unica scelta possibile anche se, avendo usato un ORM come SQLAlchemy, è probabilmente possibile usare la stessa classe DbFetcher per ogni tipologia di DBMS. Idealmente è possibile scrivere nuovi fetchers anche per risorse che siano diverse da database (file, streams, ..). Sull'oggetto fetcher viene invocato il metodo "fetchBetweenDates" che esegue una query remota ed esegue "magicamente" il mapping tra le colonne remote e i campi della classe interna del metaparametro corrente (es: per "MetaExample1" la classe interna è "Example1RemoteMapped"). Infine i dati ottenuti vengono normalizzati e trasformati in serie temporali come introdotto in precedenza (metodo "fillTimeSerie" delle classi dei metaparametri).
6. persistOnDb. Per ogni parametro scelto, viene svolta una query che carica, per ogni suo intervallo temporale trovato nella tabella readElement e compreso nel range impostato, solamente le righe a versione massima. In tal modo si ha in memoria una lista di readElement precedenti "attivi" (non sovrascritti da versioni più recenti) relativi allo stesso parametro e agli stessi intervalli della query. A questo punto viene svolto un ciclo che itera su ogni intervallo temporale del range e calcola un nuovo readElement, con eventuale decomposizione geografica (chiamando il metodo "calculateValue" del parametro), passando ad esso la porzione riferita all'intervallo corrente della serie temporale caricata. Ciascun nuovo readElement creato avrà versione massima per il proprio intervallo. Una volta scansionato tutto il range si procede con il parametro successivo. Nella seguente immagine viene mostrata la tabella readElement al seguito di una seconda esecuzione del servizio relativo ad un esempio già introdotto.

idreadElement	readData	startTimeStamp	endTimeStamp	version	user_iduser	parameter_idparameter
1	0.502836	2015-05-01 00:00:00	2015-05-01 23:59:59	1	NULL	2
2	0.491533	2015-04-30 00:00:00	2015-04-30 23:59:59	1	NULL	2
3	0.502836	2015-05-01 00:00:00	2015-05-01 23:59:59	2	NULL	2
4	0.491533	2015-04-30 00:00:00	2015-04-30 23:59:59	2	NULL	2
NULL	NULL	NULL	NULL	NULL	NULL	NULL

Interfaccia ed effetti del servizio di Run (RunModelService)

Analogamente a quanto visto per il servizio di fetch, anche il servizio di run è "pilotato" sulla base di alcune impostazioni che vengono chieste all'utente (ov_service_runmodel_main.py). Anche qui, per la creazione del servizio, sono richiesti all'utente vulcano di interesse, elicitazione e mappa. Inoltre durante la chiamata di run è richiesto anche il nome di un runModel, ossia di un modello di calcolo generico scritto da uno scienziato. Tale nome deve essere presente nella tabella runModel (attributo "className") e deve corrispondere ad esso anche una classe con stesso identico nome in un file della sottodirectory runmodels (è già presente l'esempio "TestModel"). Il metodo "run" del servizio prende in input tale nome ed un campo che rappresenta l'email di un utente registrato. Quest'ultimo dato serve solamente nel caso in cui il servizio viene lanciato manualmente da un operatore.

```
runmodelService.run(runModelClassName, None)
```

Il metodo "run" ha come riferimento il timeStamp attuale per costruire i dati inerziati di input. E' anche possibile lanciare l'esecuzione su un istante passato in modo da avere "istantanee" del modello in tutta la sua storia. Per fare ciò è sufficiente aggiungere un terzo campo con la data passata.

```
runmodelService.runOverHistoricalData(runModelClassName, None, datetime.datetime.strptime("2015-5-05
0:0:0", "%Y-%m-%d %H:%M:%S"))
```

Provando a lanciare l'esecuzione di quest'ultimo metodo, tenendo in mente gli esempi riportati in precedenza, si ottiene nella tabella inputElement il seguente risultato:

idinputElement	value	parameter_idparameter
5	NULL	1
6	0.497184	2
7	NULL	3
8	NULL	4
NULL	NULL	NULL

Le 4 righe sono relative ai 4 parametri presenti nella tabella parameters (in quanto tutti appartenenti a stessa elicitazione); questi sono "example1", "example2", "upliftthreemonths", "tremortwomonths". Essi sono tutti nulli escluso "example2" in quanto, come osservato, la tabella readElement contiene valori solamente per tal parametro. In particolare il valore calcolato è la media dei dati relativi ai 10 giorni precedenti al 5 maggio (data passata a cui si è riferita l'esecuzione). Essendo nella tabella readElement presenti dati solamente del 30 aprile e 1 maggio la media è calcolata su questi due valori. La scelta del calcolo della media sull'arco temporale di 10 giorni è stata descritta nel codice della classe "Example2".
Analogamente, essendo "example" un parametro georeferenziato, ed essendo stato fornito il codice per il calcolo del valore inerziato sulla mappa, nella tabella inputElement_decomposedIn_geographicSample vengono riportate le medie sui 10 giorni della decomposizione geografica del parametro.

idinputElement_decomposedIn_geographicSample	geographicSample_idgeographicSample	inputElement_idinputElement	value
1	701	6	0.497333
2	702	6	0.494138
3	703	6	0.506059
4	704	6	0.5255
5	705	6	0.479675
NULL	NULL	NULL	NULL

Nella tabella run viene scritta la data di esecuzione, Id del modello di calcolo (in questo caso "TestModel"), Id della mappa (in questo caso "CardinalModelTest"), l'utente che ha lanciato la run, (solo caso manuale) e un campo di output "libero", in cui ogni modello di calcolo può scrivere il suo output peculiare. Si noti che la data scritta nella tabella di run NON corrisponde al 5 maggio, data usata come query per il calcolo dei valori di input, ma alla data odierna. Questo significa che i dati di input non sono ricostruibili; questa scelta è stata fatta in quanto, siccome i dati remoti potrebbero cambiare anche in date passate, è preferibile ricalcolare il modello con dati recenti (fetch + run) che guardare tabelle di vecchie esecuzioni non più affidabili.

idrun	timeStamp	output	user_iduser	runModel_idrunModel	mapModel_idmapModel
2	2015-05-16 11:29:00	Run OK	NULL	1	2
NULL	NULL	NULL	NULL	NULL	NULL

Una volta preparati i dati in input, il servizio di run trasforma tali dati per renderli facilmente accessibili dalle classi di modello di calcolo. Qui di seguito è introdotta la classe astratta (sottocartello runmodels) da cui tutti i modelli di calcolo devono ereditare. Come si può notare durante l'inizializzazione viene passata una lista "inputParameters"; questi sono oggetti di tipo "InputParameter" formati ciascuno da una descrizione del parametro a cui è riferito (soglie,

relazione, peso, ...) più un valore (calcolato dal servizio di run) e la sua eventuale decomposizione geografica nelle aree della mappa. In questo caso la lista lista "_geographicDecmposition" è di N valori, dove N è il numero di campioni geografici della mappa. I valori sono ordinati; quindi gli indici della lista corrispondono agli indici del campo "index" della tabella geographicSample. Volendo è quindi possibile risalire a dati quali la struttura geometrica di un certo campione. In input al modello di calcolo viene fornita anche la mappa, da cui è possibile estrapolare informazioni sui campioni stessi.

```
class AbstractModel(object):

    def __init__(self, inputParameters):
        self._inputParameters = inputParameters #see InputParameter class
        self._mapModel = None                   #mapModel object. It can be useful to give an inter-
pretation to  _geograpicDecomposition. mapModel.gegraphicSamples indces can be used to access (by
index) to _geograpicDecomposition list, and have spatial informations about the model.
        self._output = None

    def run(self):
        raise NotImplementedError("This class is to be considered not-instantiable.")

    @property
    def output(self):
        return self._output

    @property
    def mapModel(self):
        return self._mapModel

    @mapModel.setter
    def mapModel(self, mapModel):
        self._mapModel = mapModel


class InputParameter(object):

    def __init__(self):
        self._description = None #string
        self._value = None       #float   (inertia-value (like avg of last 90 days)
        self._threshold1 = None  #float
        self._threshold2 = None  #float
        self._relation = None    #string
        self._weight = None      #int
        self._nodeName = None    #string
        self._nodeIndex = None   #int
        self._geographicDecomposition = None  #list of float values. To give a correct interpreta-
tion of the map structure used to calculate this list you should use mapMopdel
```

La seguente classe "TestModel" è ereditata da "AbstractModel" e l'unica cosa che deve fornire è un metodo di run e scrivere nel campo "_output". Sarà il servizio di run a preparare i dati in input e lanciare il run di questa classe; lo scienziato è quindi sollevato da qualsiasi compito che non sia scrivere il modello all'interno del metodo di run.

```
#class name must match the "className" attribute of runModel table
class TestModel(AbstractModel):

    def __init__(self, inputParameters):
        AbstractModel.__init__(self, inputParameters)

    def run(self):

        for inputParameter in self._inputParameters:
            #Do stuff with current inputParameter
            #...
            #...
            a = 1

        self._output = "Run OK"
```

Una volta terminato il calcolo, il valore della variabile "_output" verrà scritto sul database nella

tabella run dal servizio. In futuro è quindi pensabile ad un ulteriore servizio in grado di interpretare i dati presenti in questo campo e stamparli a video.

Implementazione del servizio di Run (RunModelService)

Di seguito sono riportate le descrizioni dei passi implementativi per raggiungere i risultati mostrati.

1. InitRunEntity. Vengono caricati in memoria mappa, utente (opzionale), runModel e parametri. Per ogni parametro vengono letti, per ogni parametro, i readElement a versione massima (corredati di decomposizione se disponibile) relativi agli intervalli temporali compresi nel periodo di inerzia del parametro stesso (es: 90 giorni per "example1", 10 giorni per "example2"). A questo punto viene creata una serie temporale con la lista dei readElement letti. Se è disponibile anche la decomposizione su mappa, vengono preparate altre N serie temporali, una per ogni campione geografico. Con i dati così organizzati è possibile chiamare il metodo "calculateInertiaValue" del parametro corrente e costruire con il risultato l'inputElement e la eventuale decomposizione inputElement_decomposedIn_geographicSample. Svolta l'operazione per tutti i parametri si possono scrivere sul database questi risultati linkandoli ad una nuova riga della tabella run.
2. LaunchRunModel. I dati creati devono essere elaborati per renderli disponibile in un formato richiesto dagli scienzati responsabili della scrittura dei modelli di calcolo. Per fare questo si creano tanti InputParameter quanti sono i parametri (vedere sottocartella runmodels). Per ognuno di essi vengono scritti campi che sono relativi alla descrizione del parametro, al suo valore appena calcolato e alla decomposizione di questo se presente. Creati i InputParameter il servizio passa ad eseguire un parsing del nome della classe del runModel corrente ed istanzia un oggetto relativo a tale classe passando ad esso i InputParameters appena creati insieme alla mappa. Finalmente si è pronti a lanciare il metodo "run" dell'oggetto creato; qui dentro lo scienziato può scrivere la sua analisi dei dati e produrre un output.
3. SaveOutput. L'output creato alla fine della procedura viene scritto nella riga relativa dell'oggetto run creato alla fine del passo 1.

Simulatore di test, un tool per lo sviluppo

Ai fini dell'implementazione dei servizi introdotti si è reso necessario progettare in parallelo uno strumento in grado di creare database remoti con dati simulati. In questa maniera è stato possibile effettuare test in vitro senza avere accesso ai database reali. E' già stato presentato un loro utilizzo all'interno dello script di inizializzazione.
Analogamente a quanto visto per il database locale anche per la costruzione di questi database di test si è sfruttato SQLAlchemy per mappare delle classi in tabelle. In particolare nel file test_remote_model.py sono presenti due classi: SimulatedObservation1 e SimulatedObservation2. La prima viene usata per rappresentare parametri georeferenziati (valore, timestamp, coordinate X e Y), la seconda per parametri di cui è noto solamente valore e timestamp.
All'interno dello script test_remote_simulator_main.py è presente un main in grado di generare un database remoto della struttura definita, come introdotto, in test_remote_model.py (lo stesso codice è stato portato anche nel processo di inizializzazione presentato).